SETL Newsletter # 149

J. T. Schwartz
April 20, 1975

Conventions allowing other languages to be used within GYVE; Files; Memory hierarchy questions; Some suggestions for GYVE extensions.

1. Use of other languages.

GYVE as presently specified pretends to be a monolanguage system, though of course as a practical matter this restriction against using other languages is unacceptable. The following quite simple suggestion removes this restriction, making other languages, and as a matter of fact machine language, available.

i. two new types of GYVE objects, the *binary program* and the *binary program pointer*, and a new statement

(1) FORMPROG(bits) IN(account) SET(bpointer)
<parameter-list>

or

(1') FORMRENTRANTPROG(bits) IN(account) SET(bpointer)
<parameter-list>

will be introduced. Here, *bits* designates an array of bits, giving the machine (or other interpretable) code constituting the program, plus appropriate areas for any static local data associated with this code. The <parameter-list> appearing in (1) consists of consecutive *parameter-declarations*, each having the form

(2) NAME(parameter-descriptor) AT(n).

In (2), *parameter-descriptor* is much the same as a GYVE parameter descriptor, except that only integers and specified-size bitstrings, character strings, and arrays of these objects, but not pointers, may appear in (2); and n is an integer. The clause (2) informs the GYVE system that a binary representation of the parameter described by (2) appears, in a standard (but implementation-dependent) form, in the data area of the binary program B declared by (1), stating at data address n. (if the program is not RENTRANT, data addresses are the same as binary addresses. The binary program B is free to read and to modify all of its own data area any time it is executed; in accessing one of the parameter areas defined by (2), B may of course wish to make use of standardised 'library routines'.

ii. Binary program pointers (and pointers to processes formed using binary programs, see below) satisfy the normal GYVE rule that an object O may only be assigned to a variable giving no more privilege in the use of O than the variable from which O is retrieved. Note that by omitting parameters of a binary program we diminish privilege.

iii. A process PR (known by the GYVE system to incorportate 'foreign' code) can be formed from the program PG declared by (1) using the ordinary GYVE 'create' mechanism. The INITCALL statement which is then required before PR can execute is then used, in much the ordinary way, to initialise the parameters appearing in the <parameter-list> of (1). If PG has been declared RENTRANT, then PG need not be copied when PR is created; otherwise, creation of PR will involve the formation of a fresh copy of PG.

PR will always execute under hardware constraints which prevent it from writing outside the memory area allocated to it. Within this area it may of course write arbitrarily and unpredictably.

Given a pointer PPR to PR, we can write into a parameter called NAME of PR by writing SETPARM(PPR.NAME)TO(expn)RESULT(R); if NAME is an array, this should be SETPARM(PPR.NAME(index)) RESULT(R). This same parameter can be read by writing

FETCHPARM(PPR, NAME(index)) SET(V) RESULT(R).

iv. A period of execution of PR can terminate in one of three ways: by timeout, by error signal, or by service request (perhaps finish request). The type of termination that occurs will be signalled in the normal way in the RESULT parameter of the EXECUTE statement which activated PR.

Termination by error will occur if fault is detected during execution of PR. Faults can be either out of range faults, or other hardware-detected faults such a divide-by-zero. The detailed consequences of a fault are implementation-dependent, but will generally be recorded in bits of an obligatory parameter of PR named FAULT; this parameter, since it is obligatory, should not be described in the <parameter-list> of (1). The precise layout of the FAULT parameter is implementation dependent; information in rationalised form can of course be retrieved by calling appropriate implementation-dependent routines and transmitting the FAULT parameter.

Termination by service request will be available through some type of emulated or hardware supervisor call instruction. The precise service being requested will signalled by the settings of conventional parameters of PR. Service routines may then modify the values of other parameters of PR, signalling the outcome of the requested service in some other agreed-upon way, which will leave PR in a self-consistent state, ready to resume or continue execution.

2. Some suggestions for GYVE extensions.

a. Treatment of 'RESERVE'.

The GYVE 'reserve' construction, needed on the one hand to prevent important global instances from becoming inadventently or maliciously blocked, can lead on the other hand to an unfortunate accumulation of time granularities, which can ultimately destroy the ability of GYVE-written systems to respond rapidly in real time. The following suggestion, which does not violate the principle of 'nesting' implicit in GYVE, is intended to alleviate this difficulty.

i. Each process will carry an additional parameter called its *rlevel*, which will be assigned to the process at the time of its creation. This value can be supplied as an additional argument n in the basic process create statement; n must be non-negative, and the process being created will have an rlevel n more than the rlevel of the process creating it. If n is not explicitly supplied, it defaults to 0.

ii. If an instance I carries a RESERVE clause, this clause will carry an rlevel parameter RL in addition to the 'time' parameter TR presently specified. The parameter RL will be assigned at the time that the instance is created; the rule just explained in connection with process creation will apply.

The additional parameter RL will be used as follows: if a process P_k sitting at the end of a chain $P_1, P_2, \ldots, P_k$ of executes attempts to enter I, entrance will be refused unless the process P_j in the chain which will time out first has an rlevel parameter smaller than or equal to RL. But then P_j is allowed to time out in the normal way even before return from I is made.

iii. After I has been entered, before TR time-units have expired, and before return is made from it, no process P_j in the chain $P_1, P_2, \ldots, P_k$ which has an rlevel parameter larger than RL can gain control through the receipt of a signal at a port. Signals which would have this effect are merely stacked at their ports of receipt.

Note that the mechanism which has been described allows a high-level initiator process P_1 to regain control immediately on the occurence of certain real-time-significant events, even if a process directly or indirectly executed by P_1 has entered an instance for which a RESERVE is posted.

An example of the use of the mechanisms just described will clarify their intent. Suppose that a process P_o is scheduling the execution of other processes $P_1, \ldots, P_n$. P_o may want to allow $P_1, \ldots, P_n$ to communicate via some instance I, accessible to all of them, but created by P_o. The instance I may also be accessed by processes created and scheduled by any P_j. Now, I can be coded to protect itself from all misuse, except that if I is entered by a process PS subordinate to P_j which then times-out, returning control to P_j, then P_j can inadvertently or maliciously refuse to reschedule PS, thus hanging I.

To prevent this, P_o must be able to prevent control returning to P_j, either on a timeout or through an interrupt, after PS has entered I and before it has returned from I. The existing reserve mechanism will do this, but at the cost of making it impossible for P_o to time out or receive an interrupt while PS is executing I. The mechanism just described relaxes this unduly severe restriction, by allowing P_o to give I a reserve level of n, while all the processes P_j have reserve levels of n + 1 at least.

b. 'Inspect' Entries.

GYVE presently distinguishes between 'reader' and 'writer' entries to an instance I; when I has been entered at a 'writer' entry by a process P, no other process is allowed to enter I, even to read values of its interally stored variables. The rationale for this rigorous exclusion is that these variables would be changing in ways confusing to other readers as P executes. However, there do exist certain types of secondary processes for which this argument is irrelevant, e.g., 'spy' or 'monitoring' routines which wish to inspect P's environment periodically to estimate P's progress and/or statistical behavior. For this reason, it may be worth introducing a third class of instance entry, the 'inspect' entry, which gives read-only access to the instance's internal data, but which can be entered even if a write entry to the same in- has already been invoked.

3. Memory Hierarchy Questions.

GYVE as presently defined incorporates no mechanisms for manipulating the residence level of code or data; a 'single level' store is assumed. This approach is valuable for the conceptual simplifications which it affords; at the same time, however , it can create practical difficulties for GYVE-written systems.

a. A 'single memory level' system will rest, at the implementation level, on some paging scheme. GYVE does not provide any means for responding to the 'page-thrashing' situations which can arise in paging systems.

b. If a GYVE system is to be supported by hardware which does not include any memory mapping scheme, support of a 'single level' store model may make it necessary to use software paging schemes. Schemes of this kind may be inefficient; still worse, if foreign software products are to be made available using a method like that suggested in Section 1 above, they may be impossible.

c. Global objects, especially files, which have remained inactive for substantial periods of time should either be pushed to archival storage levels or purged. GYVE does not provide any mechanisms either for locating or for unloading such objects.

Note also that we will want to use the GYVE coordination and exclusion mechanisms to organise both highly dynamic collections of interacting processes and slower-moving files, which may be large and resident principally on secondary storage devices. It is of course important that the heuristic notion of file be modeled in a way which is both helpful and rather directly based on the mechanisms presently available in GYVE, since if this is not the case it will become necessary to invent entirely new file management mechanisms, which would force one both to rethink some of the central problems of coordination and control which GYVE already addresses, and to accomodate, to the existing GYVE framework, whatever solutions were found.

Found with the problem listed above, we make the following suggestions for adapting GYVE to a hardware situation in which a fast central memory is backed up by several large storage disks (and possibly by some amount of 'bulk electronic memory' also.)

i. In addition to GYVE 'accounts' of the present type, central memory accounts, which will be called *cmaccounts*, will be introduced. (If several levels of memory hierarchy were to be distinguished, then account types corresponding to each such level might be introduced. However, we shall for simplicity begun by ignoring this extra possibility). *cmaccounts* can, like ordinary GYVE accounts, be allocated within each other; collectively, cmaccounts form a tree. Essentially, a cmaccount represents a block of central memory, available as'paging space' to processes which have a pointer to it.

ii. A process P which is to be executed must always be 'attached' to some cmaccount. This 'attachment' will be set up by a statement

```
ATTACH(P) TO (CMACT);
```

which must always be invoked before P is first executed. Then some suitable 'topmost' portion of P's invocation record stack, plus all instances paged in by P, will be held in the cmaccount to which P is attached. The essential rules governing paging are as follows: if an object which P attempts to use is already in central memory, it will be used no matter what cmaccount contains it; but if it is initially present in central memory so that paging is necessary, the cmaccount A to which P is attached will be used, which may mean that some other data object is paged out of A. Note that only a process which is attached to a cmaccount A can cause anything to be paged out of A (by calling for something else to be paged in.)

iii. Pages of some nominal size will be provided. The whole of an instance may be resident on one 'logical' page LP (so that if the instance is large LP may be many times the minimal page size.) It is also possible for an instance to be 'internally pageable', allowing it to be entered and to execute even if all of its parts are not physically present in central memory (conventions supporting this extended facility will be described in more detail later.)

iv. When a page fault occurs, a paging operation will be initiated. This situation will be seen by the GYVE system as a 'block' (of the instance executing at the moment the page fault occurs), followed by a 'wakeup', which will occur when the required page has been brought in. Responsibility for scheduling other work during the paging operation belongs to the process to which the block gives control.

v. If the cmaccount to which a process P is attached is erased, the process will not be able to execute, and will experience a fault. We allow several processes to be attached simultaneously to one single cmaccount. A process P can be detached from its cmaccount A by executing the statement

DETACH(P).

Such a detach does not destroy P, but merely 'rolls it out' to secondary memory; the same rule applies if A is erased while P is executing. If P is detached while executing, a fault will be experienced. If P attempts to page in an object which is too large for its cmaccount, a fault will also be experienced.

In a hardware environment not providing hardware paging, the GYVE compiler will probably support software paging. This will make it unnecessary for the whole of a large instance to be brought into core before the instance can begin to execute.

Some suggestions concerning software paging techniques which might be used will be found at the end of the present newsletter. Note however that the whole body of a process incorporating non-GYVE binary code may have to be brought in before the process can begin to execute.

vi. If a process P accesses a record of a FILE aggregate F (see section 4, below) then either the record as a whole or some paged subpart of it will have to be brought into the portion of central memory belonging to the cmaccount to which P is attached. Fragments of the index through which F is addressed may have to be brought in also. These index fragments will be retained and used to expedite reference to successive records of F.

vii. Certain important scheduling decisions, including a scheduler's decision as to whether to increase the number of programs among which it is multiprocessing, will depend on the paging rate experienced by the various central memory accounts which P is supervising. To make this available, we provide each cmaccount A with an entry called PAGERATE. The call A.PAGERATE(R) sets R to some fair representation of the paging rate recently experienced within A.

viii. If available memory actually consists of several hierarchical levels of storage through which information must be paged, then the 'cmaccount' concept described in the preceeding pages can be generalised to provide accounts at different levels. To execute, a process would then have to be attached to an account at each hierarchical level; when an account at one level was destroyed, the global objects in it would automatically be rolled out to the next lower level of memory. Note that an account at any but the bottom memory level would always be part of a larger, lower-level account.

4. Files.

GYVE presently treats the semantic notion of 'file' as being identical with that of 'array of (fixed format) records'. Several objections can be raised to this model of the 'file' concept:

a. It is not uncommon for the records of files to differ widely. For example, files whose records can in principle hold dozens of different information fields, but where in practice almost all of these fields are missing, often occur.

b. One will often want to keep files in a sorted order permitting fast binary search. One wants it to be possible for new records to be inserted into the file, and for old records to be deleted, without extensive physical reorganization becoming necessary, and without old record indices becoming invalid.

c. The fields on which the records comprising a file are sorted and searched deserve to be treated somewhat differently from the other fields of the file's records; in particular these 'key' fields should for fast search be held in physical proximity to the actual indices which locate the records of the file.

d. 'Key' fields of files need not be unique.

For all of these reasons, it is suggested that a more appropriate model of the concept of 'file' is that of a (possibly multiple-valued) function f defined (perhaps sparsely) on a totally ordered set s'.

It may of course be undesirable to introduce any specific features into GYVE simply in order to model some specific file concept. It could be argued that no one 'access method' will be entirely satisfactory, so that file models, like scheduling algorithms, should simply be left open 'for programming'. Against this, several points can be argued. First of all, the programming needed to support an effective file system is not simple. Multi-way balanced

trees are an attractive implementation-level structural form; these are complex enough so that a programmer, left to his own devices and unable to get at the lowest-level mechanisms of an implementation, could easily come up with a personal file system that was both slightly misconceived and considerably short of best attainable efficiency. Secondly, the present version of GYVE provides no easy way in which multiple parts of a data aggregate of dynamically variable structure, held together by auxiliary indices, can be used simultaneously by independent processes. For both these reasons, we shall go ahead to propose particular file mechanisms for GYVE. Of course, a more general solution to the linked problems of data aggregation and parallel use of aggregate subparts, based perhaps on some notion of 'programmable' as distinct from merely 'static' pointers (of the kind now provided by GYVE) might be more desirable over the long term, and might make it possible to avoid commitment to any one particular file model.

But to proceed: The general approach to the 'file' concept suggested by our earlier paragraphs leads to the following linguistic conventions and implementation techniques.

i. An aggregate mode of type file is declared as

(1) *modename:* FILE *keyfields-declaration-part*
RECORDS *record-declaration-part.*

Here the keyfields-declaration-part is a list of variable declarations, each variable being either an integer, a bitstring, or a character string. Variables declared in the keyfields-declaration-part of (1) will be used to sequence the file, and will be held in appropriate physical proximity to the master index through which the file is accessed. The file will always be sorted into the (lexicographic) order determined by these variables (whose lexicographic priority is determined by their order of declaration).

ii. As a nicety, we allow the record-declaration-part of a file declaration to consist either of a single *record-declaration* or of a sequence of record declarations, each prefixed by a *kind-name*. Examples would be

α. single record declaration.

```
WEIGHT:INT
AGE:INT
```

β. sequence of record declarations:

```
KIND(FARM)
   ACRES:INT
   INHABITANTS:INT
   TOWNSHIP:CHAR(50)
KIND(APARTMENTBUILDING)
   INHABITANTS:INT
   TOWNSHIP:CHAR(50)
   ADDRESS:CHAR(50)
   FAMILIES:INT          , etc.
```

iii. As seen in the preceding example, the record declarations occurring in a record-declaration part consist of a sequence of variable declarations of the kind already provided in GYVE. As an additional nicety allowing compression, we allow a list of 'likely' values to be appended to any one of these variables. This list is preceded by the keyword LIKELY; for example, we may write

```
COLOR:CHAR(10) LIKELY(RED,GREEN,BLUE) .
```

Likely values will be encoded using compressed bit patterns, making it unnecessary to store the full representation of a likely value, which can achieve significant data compression. LIKELY values can also be stated for an array of scalar quantities, in which case every array component having a likely value can be represented by just a few bits, allowing even more significant compression. Still more general mechanisms allowing likely values for structures, and for the components of arrays of structures,can be defined and

will allow files to be compressed still further.

iv. As noted above, we allow any number of records within a file to have their key field(s) in common. To make record references unique, we therefore attach an additional, implementation-level field to each record of a file; we call this its *supplementary field*. The contents of this field are generated by the GYVE system at the time that the record in question is inserted into the file; certain of the file access primitives to be described below will require, and others will supply, a supplementary field value.

v. Access to the records of a file F is tained by an instance call whose basic form is illustrated by

(2) F(K) .FARM. ACRES SET(XACRES) [RESULT(R)][NEXTK(K',XK)]

Here, K is a key to the file F (which for simplicity we suppose to have only one single key field); FARM makes definite the class of record expected, and ACRES extracts a field of that record; and XACRES is a variable into which the value retrieved by (2) is to be placed. If the optional result parameter R is supplied, then 'no record with this key' may be signalled through it.

If in F there exist several records with the key K, then the call (2) will access that one of these records which is 'first' (in an implementation-defined sense). If the optional parameters K' and XK' are supplied, then on return from the call (2) they will respectively have been set equal to the key and the supplementary field value which locate the next record (in logical file sequence) after the field addressed by (2). To make it possible to call, using this supplementary field value, for some particular one of a group of records all of which share a common key, we allow (2) to appear in the modified form

(2') F(K,XK) .FARM.ACRES SET(XACRES)[RESULT(R)][NEXTK(K',XK')].

The form (2') differs from (2) only in that both the key K and the supplementary field XK appear in parentheses following F; which singles out a particular one of the records sharing the key K.

vi. One of the entries E which can appear in a call (2) or (2') is 'DELETE',which if available and invoked will delete the record instance it references from the file F. A related entry, INSERT, having the general form

(3) F. INSERT(K).RECORDKIND SET(XK') [RESULT(R)]

will be provided. This will insert a new record with key K and of the specified RECORDKIND into file. The field XK' will be set equal to the supplementary field value which uniquely distinguishes the newly inserted record. This record will be initialized in the manner specified in the declaration of the file mode to which F belongs.

vii. Note, as an implementation-level matter, that the supplementary field value XK' returned by a call (2') or (3) can contain a pointer to the file index fragment through which the record called for by (2') or (3) is accessed. This information can then be used to minimize the number of physical accesses required to access the records of a file if the pattern of accesses is serial.

viii. It may be desirable to allow optional parameters PREVK(K',XK') in a call (2'), making it possible for files to be accessed in 'backwards sequential' order.

ix. Some processes will only require exclusive access to the individual records of a file; others will need to have exclusive access to the file in toto. This can be handled without introducing any new feature into GYVE. The following approach, which is also valid for aggregates more general than files, and which is of acceptable though not excellent efficiency, can be used. If a file F may have to be reserved *in toto*, pass the actual pointer PT to it through

a valve, and pass the resulting valved pointer VPT, rather than PT itself, to processes which will only need to access individual records of F. Then give processes needing to reserve all of F access to an instance I able to reference the valve V through which PT has been passed. When I is entered, it will shut the valve V. Then, using a 'semaphore instance' counted up whenever access to a record of F is accessed and counted down whenever this access terminates, I can wait for all in-process accesses to terminate, and can then go on to access F. After return from I the valve V should be opened again. (Means for ensuring that this will be done are desirable.)

x. A process which enters a mode instance, and especially one which enters a record within a file in order to read or write part of the internal data of the instance may in some cases depend on the assumption that the data passed satisfies certain consistency conditions. For this reason, it may be unacceptable for this data to be accessed piecemeal via a series of separate entries. To make several successive accesses unnecessary, the following construction, tentatively proposed, may be useful.

a. Introduce the notion of a 'substructure' or 'subrecord' SS of a GYVE structure S. This can be defined as follows: If S is a record with named subfields, then SS will have named subfields; the names used must be among those which occur in S, and the items in SS which they name must recursively be substructures of the identically named items in S. If S is a scalar item, then SS must be a scalar item of the same kind, or be void. If S is an array, then SS must be an array or an array component, and its declaration will include a clause of one of the forms

TAKE(expn$_1$,expn$_2$) (for subarrays)

or

TAKE(expn$_1$) (for components).

Here, *expn*$_1$ and *expn*$_2$ are integer valued expressions, which

delimit the subinterval (first case) or the component (second case) of the array S to which SS refers.

b. If A and AA have been declared to be variables of the types S and SS respectively, permit assignments

AA = A and A = AA,

having something of the force of PL/I 'by name' assignments. The first of these will extract all the fields, components, and subarrays constituting AA from the similarly named fields and appropriately delimited subparts of A; the second will set the fields of A named in AA and the parts of A delimited in AA from the values which AA contains.

c. Permit assignments of the type just described, even if AA, A or both are parameters of the instance in which the assignments occur. Moreover, do this dynamically, in the following sense: allow the actual parameters with which an instance entry is called to be either substructures of the corresponding formal parameter of the entry, or to be superstructures of the entry. When this occurs, make up a 'dope vector' giving all actual details of the argument-to-parameter correspondence which prevalls for a given call. Then assignments to a formal parameter can be used to charge only some part of the actual argument, and formal parameter retrievals may fetch only some *relevant* subpart of an actual argument.

Note in connection with all of this that the features just described are harmless, in that they can be implemented, albeit with considerable extra user-level programming, in the existing GYVE system. (To do this in the case of instance arguments for which assignments A = AA etc., must be handled dynamically, it would be necessary to make up a 'dope vector' of appropriate form, and to pass this as a parameter along with A or AA.) However, the features described can be quite useful, in tha they may make it convenient to trigger actions whose description would otherwise be long and clumsy.

5. Detecting and erasing or archiving inactive objects.

If some global object of a GYVE system remains inactive for a substantial period of time, it may be desirable to erase it, or perhaps to move it to some 'archival' storage medium such as magnetic tape. To this end, the following mechanisms are proposed.

a. A proccess having a pointer to an account A (and hence able to destroy the account *in toto*) will be able to execute a call

(*) A. INACTIVE(date) NUMBER(n) [ARRAY(ar)].

In this call, *date* is an integer representing some prior calendar date; it is objects belonging to the account A, and not accessed since *date* that the call (*) is to return. Pointers to up to n of these objects are to be placed in the array *ar*; if fewer than n inactive objects are found in A, then n, which is an integer variable, is to be set equal to the number of these inactive objects. If more than inactive objects are found, then n can be changed to n + 1.

b. Once pointers to the inactive objects of A have been collected in *ar*, these pointers can be used to erase any or all of these objects. Alternatively, all or some of these objects can be packeted, and the resulting packet archived, perhaps by being written to tape.

c. In a system providing for the archiving of inactive objects, such objects will generally be archived without the knowledge or intent of their would-be users, who may appear at a later date, discover that these objects have been removed from the system, and wish to restore them. To this end, we propose the following mechanisms.

i. If a process blocks because it tries to invoke an erased or archived instance, make some appropriate external symbolic form of the pointer instance available, say via an auxiliary process entry called

BLOCKREASON.

ii. Make the function which converts a pointer to its external symbolic form independently available. Then, when objects are purged or archived, add the symbolic forms of their pointers to overall system lists of purged and/or archived objects.

iii. Provide an operation which can read an archive tape and restore a copy of an object with (symbolically) specified pointer to any account which is large enough to hold this object.

Note however that no operation converting a symbolically specified pointer to an actual GYVE pointer will be provided. Thus pointers remain 'unforgeable', and in particular no process not able to access an instance originally can access the object after it has been archived and restored.

6. Preliminary remarks on reliability and recovery issues.

Perhaps the worst exposure of the GYVE system lies in its assumption that its hardware and software will always work as specified, and in the lack of any attention to the problem of recovery from dynamically detected errors or after system crash. In the present section, we will try to say something about these important issues.

We model the problem of recovery after crash as follows: The state of a GYVE system is unpredictably modified by subjecting the implementation-level bit patterns representing some small fraction of the many global objects which the system contains to random change, and by destroying all or part of the master index which locates global objects given their pointers. However, the bulk of the large data base stored by the GYVE system survives. The problem is then to restore GYVE to a 'runnable' state, with automatic rebuilding of as many global objects as possible, and to facilitate the restoration, perhaps from backup copies, of objects hopelessly destroyed. In this latter connection, some systematic way of archiving backup copies of newly created or modified objects, and of journaling important symbolic input, may be desirable.

Note that a recovery procedure will, generally speaking, resemble a startup procedure, except that a startup procedure will set up an initially empty family of accounts, whereas a recovery procedure will be confronted with some pre-existing collection of accounts, and will be concerned to preserve all objects held in these accounts.

In overall outline, we propose the following approach to the problem of post-crash recovery.

a. The physical representation of each global object O consists of some collection of physical 'pages'. Each such page will be checksummed, probably when the page is written out to central memory; enough different checksums will be used to make it highly unlikely that a page with valid checksums actually contains a fault. Each page p of O will also carry a page number, which locates p within the group of pages representing O, a copy of the global pointer which identifies O, and a field defining the account to which p belongs.

b. If the object O is a file, its records will be laid out on pages in a way permitting the reconstruction of every record not stored in whole or part on a corrupted page, even if some of the pages used to represent the file are lost. This allows the file to be reconstructed, with loss of only a few records, even if a few of the files pages are lost.

c. Recovery, in as complete a form as possible, of certain particular types of objects is particularly important. Among the items whose recovery is critical, we note the following:

i. Most or all of the tree of accounts needs to be recovered. Recovered global objects need where possible to be assigned to the same accounts which held them before crash.

ii. User 'personal catalogs' or 'holder instances' need to be recovered where possible. It is through these catalogs that a user is able to connect to his programs and files. 'User table' entries containing user passwords, pointers to personal catalogs, and possibly also some small amount of accounting and user profile information, also need to be rescued.

d. To preserve particularly significant global objects we propose to create multiple copies of them at physically different storage locations. For this, we propose to allow a clause

INSURE(n)

to be attached as a qualifier to the definition of a mode. When return is made from a non-reader entry of an instance of the mode, n copies of the instance will be created, all in different storage locations. These copies will be chained together by pointers, so that when one is destroyed all will be reclaimed.

e. Account objects will be classified as 'scratch' accounts and 'insured' accounts. These two types of accounts will behave in exactly the same way in regard to all GYVE operations, but will be handled somewhat differently at the implementation level, in ways which have consequences for efficiency, and which will also have some significance during post-crash recovery. Relevant details are as follows.

i. Any 'insured' account IA can contain subaccounts, which can either be 'insured' or 'scratch'. The total number of scratch accounts which IA can hold is limited, e.g., to not more than 63. The reason for this limitation is that the account-subaccount relationship which holds between IA and one of its scratch subaccounts SA is intended to be recoverable simply from the identifiers of IA and SA without any additional table entries being necessary. For example, IA might have an identifier XXXX00, while each scratch account subordinate to it had one of the identifiers XXXX01 thru XXXX77.

i.Whenever an account B is created within an account A, except in the case described just above, some standard number of copies of an 'account subordination record' specifying the sizes of both B and A will be created and stored in physically distinct locations.

iii. It is suggested that the following conventions be adhered to in setting up 'user tables' which the system will want to recover after a crash. In the first place, only one such table should be established within any account (which means simply that an account should be created for each such table.)

Second, the table should consist simply of a sequence of entries, each consisting of a character field holding a user name, and a pointer field which locates a standardised user information item. These user information items should be multicopied, using the 'INSURE' primitive, by the operating system procedure which sets them up. We then propose to make a primitive

COLLECT(ACCT) MODE(M) INTO(A) SET(N)

avaliable. Here, ACCT is an account, M is a mode, A is an array whose entries are of type !M, and N is an integer. This primitive will search among all the global objects contained in ACCT; pointers to the objects of mode M which are found will be placed in A. The integer N determines the maximum number of items which is desired, and is set to the number actually found; if more than N are found, N is incremented by 1.

By using this primitive one can reconstruct a 'user table' from the collection of user table items which this table originally referenced, even if the table itself is destroyed.

After a crash, a master recovery routine will be called. This will validate as many global objects as it can (by examining checksums). Objects of type 'file' will be partially reconstructed if their complete reconstruction is impossible, and some indication of the number of records lost, possibly with the keys of these fields, may be collected and perhaps spooled out to an auxiliary medium. The account tree will be reconstructed; and objects assigned to their accounts.

Note that this may involve a reduction in the size of scratch accounts from which objects have been lost. Those hopefully very few accounts whose ancestry is irrecoverably lost during the crash will be made immediate parts of a 'recovered scraps' account, which is itself an immediate part of the 'universal account' which stands at the root of the account tree. The recovered scraps account will be kept for some period of time, at the end of which it will be destroyed by master console command. Users will be supplied with a list of the accounts which have become part of the recovered scraps account , and perhaps also of the objects contained in this account; which will give them period of grace in which to form a copy of an object in the recovered scraps account in an account not scheduled for destruction. A 'lost and found user' will be created, and unrestricted pointers to all objects belonging to the recovered scraps account entered into the personal catalog of this user. By consulting the lost and found user and securing her cooperation, users will be able to recover items not otherwise available to them.

The 'logon' procedure, i.e., the procedure which validates user identifiers, will contain code allowing a user's holder instance to be called, and an account pointer AP passed back. Moreover, it will allow a user to request that the external symbolic form SYMBAPX of an account pointer be converted into the pointer APX itself, provided that the account in question is a subpart of the recovered scraps account. After APX is obtained, the account to which it points can be moved into AP, using the 'move' primitive described just below. But when this is done, the identification of the user issuing the 'move request will be written to some system file, and later printed, as a precaution against the 'theft' of items from the recovered scraps file.

In support of the operations described above, we propose the following primitives:

MOVE(APX) TO(AP) RESULT(R)

takes two account pointers APX and AP as arguments, moves APX to make it a subaccount of AP, and credits the parent account of APX with an appropriate amount of space.

FIND(SYMBPT) IN(A) SET(PT) RESULT(R)

takes a character string SYMBPT, considers it to be the external symbolic representation of a pointer p, searches in the account A for an object with this pointer and, if it is found, returns p as an unconstrained pointer to the variable PT.

Note that the FIND and MOVE primitives which have just been described can be used together to simulate the effect of a crash within an imbedded system running in some programmed 'virtual mode' within GYVE. Such a subsystem will have its own 'top level' process P, which schedules the activity of the topmost processes of the subsystem. The crash action can be programmed as a subroutine called by P on signal. It will erase some global objects, including some small number of accounts, move other accounts to a virtual recovered scraps account, and then create a post-crash recovery instance and transfer control to it. The post-crash recovery instance will then take whatever additional steps are necessary for system rebuilding. Users attached to the virtual system will experience what appears to be a real crash, and can exercise their personal recovery procedures.

An implementation note on software paging.

Software paging will always be less efficient than hardware-supported paging; nevertheless, we shall suggest a compiler-supportable scheme which should avoid some of the very worst inefficiencies which incautious software paging might imply. The scheme to be suggested will apply to multiregister machines, for which we make the 'worst case' assumption that recovery from an address-out-of-range fault is impossible.

i. A certain number of the 'index' or 'base' registers which the hardware provides will be reserved for use as 'page registers'. Every address not translated by the hardware will have to be translated using one of these registers, which means that an indexed load will be treated essentially as 'load K + XR + PR', where K is a constant, XR an index register, and PR a page register. Stores will be treated similarly.

ii. A process P will never be run unless the pages referenced by its page registers have loaded into central memory. As P runs, other pages than those referenced by its page registers can have been loaded into central memory as well, while still other pages may only be available on secondary memory. All page locations will be held in a centralised page table, from where they will be loaded into the page registers of running processes as necessary. When a load-page-register operation being attempted is found to reference a page not physically present in central memory, then a paging operation will be set in motion; this may involve unloading a page from the central memory area (account) to which the process making the page request is attached.

iii. If a process is interrupted and then resumed after a paging operation is performed, it is necessary that the contents of its page registers should be properly updated.

To ensure this, the following technique can be used:

a. The code which a process uses to search the page table can be a fixed sequence of instructions, held at a fixed location within the process. The code G should be written so that if its execution is broken off at any point and then started again from the beginning, the effect is the same as if G executes without interruption. Moreover, before it is exited, G should enter, into one or more cells associated with the page register R being loaded, a copy of the global (and hence invariant) pointer which defines the object which R will address. We shall call the cells used for this purpose 'page register content cells'.

Whenever CPU control is being restored to a process P (other than the implementation-level paging process) a check will be made to see if P was interrupted during the execution of the code G. (This only requires examination of an instruction location counter value.) If so, then P will be restarted at the beginning of G.

The paging program should adjust the contents of the page registers of any process which might be referencing a central memory area within which pages have been moved. It should be possible to do this efficiently using the page register content cells.

The following convention will allow page register contents to be moved from one page register to another without entering either system code or disabled mode: To move PR_1 to PR_2, first transfer the complement of PR_1 to PR_2, then move the global quantity held in the page register content cells associated with PR_1 to the corresponding cells associated with PR_2, and then (to signal completion of the transfer operation) complement PR_2.

This ensures that if a paging operation transpires while page register content cells are being transfered, then the code AC which adjusts the contents of page registers will recognise that the page register content cells of PR_2 are in an invalid condition. In this case, by comparing the contents of PR_2 with the contents of the other page registers, AC can determine that PR_2 is the complement of PR_1, and can therefore adjust PR_2 properly.